REMARKS/ARGUMENTS

In the Office action dated February 23, 2005, claims 1 – 3, 12 – 14, and 33 – 35 were rejected, claims 4 – 11, 15 – 22, and 36 – 43 were objected to, and claims 23 – 32 were allowed. Applicants have amended claims 1 – 43 and hereby request reconsideration of the application in view of the amended claims and the below-provided remarks.

I.   Claim Objections

Claims 1 – 43 were objected to because of informalities.

Claims 1 – 11: Independent claim 1 has been amended such that the preamble recites "A *computer-implemented* method for managing a re-usable resource" instead of "A method for managing a re-usable resource." Dependent claims 2 – 11 have been amended accordingly to recite "The *computer-implemented* method" instead of "The method."

Claims 12 – 22: Independent claim 12 has been amended such that the preamble recites "A *computer-implemented* method for managing a re-usable resource" instead of "A method for managing a re-usable resource." Dependent claims 13 – 22 have been amended accordingly to recite "The *computer-implemented* method" instead of "The method."

Claims 23 – 32: Independent claim 23 has been amended such that the preamble recites "A *computer-implemented* system for managing a re-usable resource" instead of "A system for managing a re-usable resource." Dependent claims 24 – 32 have been amended accordingly to recite "The *computer-implemented* system" instead of "The system."

Claims 33 – 43: Independent claim 33 has been amended such that the preamble recites "A computer-*readable medium* for managing a re-usable resource" instead of "A computer program product for managing a re-usable resource." Dependent claims 34 – 43 have been amended accordingly to recite "The computer-*readable medium*" instead of "The computer program product."

II.   Claim Rejections Under 35 U.S.C. 102

     A. Independent Claim 1

     Claim 1 is rejected under 35 U.S.C. 102(e) as being anticipated by Ebrahim et al. (U.S. Pat. No. 5,848,423, hereinafter Ebrahim). Claim 1 is a computer-implemented method for managing a re-usable resource. The method recites:

> "dividing a pool of integers into groups that include unique sets of integers; and
>   initializing, in computer memory, a doubly linked list that represents one of said groups of integers in response to a request for a free integer, wherein said one group of integers includes said free integer."

The support for the rejection of claim 1 is a reference to col. 12, lines 30 – 46 and Fig. 8 of Ebrahim. Applicants assert that claim 1 is not anticipated by Ebrahim for the reasons provided below.

Ebrahim does not disclose "dividing a pool of integers into groups that include unique sets of integers"

     Claim 1 recites in part "dividing a pool of integers into groups that include unique sets of integers." Applicants have reviewed col. 12, lines 30 – 46 and Fig. 8 of Ebrahim and have found no reference to "dividing a pool of integers into groups that include unique sets of integers" as recited in claim 1. In order for a claim to be anticipated, a prior art reference must disclose each and every claim element. Applicants assert that Ebrahim does not disclose "dividing a pool of integers into groups that include unique sets of integers" as recited in claim 1 and therefore claim 1 is not anticipated by Ebrahim.

Ebrahim does not disclose a doubly linked list

     Claim 1 recites in part "initializing...a doubly-linked list..." (emphasis added) A doubly linked list is a specific and well-known data structure. As defined in the reference provided by the Examiner entitled "Doubly linked lists, cursor-based lists, stacks, 5-2003, www.cs.dartmount.edu, pp.1-6," a doubly linked list is constructed from nodes which contain three pieces of data; 1) the data being stored in the list (e.g., an integer), 2) a pointer to the next node, and 3) a pointer to the previous node. This definition of a doubly linked list is consistent with the description of a doubly linked list provided in

Applicants' specification at page 11, lines 3 – 20 and Fig. 7. Additional references that support the well-known definition of a doubly linked list include "doubly linked list, National Institute of Standards and Technology, www.nist.gov" and "Linked list, http://en.wikipedia.org," both of which are attached as Appendix A.

In contrast to the doubly linked list recited in claim 1, Ebrahim discloses two separate singly linked lists. Specifically, at col. 12, lines 30 – 46 and Fig. 8, Ebrahim discloses a hash table structure (208') "in which each table entry 275' contains two link pointers, Link1 and Link2." The link pointers "are used to form linked lists of entries that have the identical hash function value." The first link pointer, Link1, for each hash table entry (e.g., slot X) "is used only when an invocation address mapping to slot X must be remapped because…" The second link pointer, Link2, for each hash table entry "is used to form a linked list (a collision list) of items mapping to the same table address as the first item in the list." As detailed in the above citations, the link pointers disclosed by Ebrahim do not define a doubly linked list. Rather, the link pointers, Link1 and Link2, define singly linked lists. Fig. 8 of Ebrahim clearly illustrates that the linked lists defined by the two link pointers do not define a doubly linked list. In particular, each node of the linked lists in Fig. 8 does not include a pointer to the next node and a pointer to the previous node, both of which are elements of a doubly linked list. Because Ebrahim does not disclose a doubly linked list as recited in claim 1, claim 1 is not anticipated by Ebrahim.

Ebrahim does not disclose that the doubly linked list is initialized "in response to a request for a free integer"

Claim 1 recites that the doubly linked list is initialized "in response to a request for a free integer." Applicants have reviewed col. 12, lines 30 – 46 and Fig. 8 of Ebrahim and have found no reference to a doubly linked list being initialized "in response to a request for a free integer" as recited in claim 1. Further, Applicants have found no reference to a "free integer" in Ebrahim. Because Applicants have found no reference in Ebrahim to a doubly linked list being initialized "in response to a request for a free integer," Applicants assert that claim 1 is not anticipated by Ebrahim.

Ebrahim does not disclose "wherein said one group of integers includes said free integer"

Claim 1 includes the limitation that the group of integers represented by the initialized doubly linked list includes the free integer. In particular, claim 1 recites "wherein said one group of integers includes said free integer." Applicants have reviewed col. 12, lines 30 – 46 and Fig. 8 of Ebrahim and have found no reference to the limitation "wherein said one group of integers includes said free integer" as recited in claim 1. Therefore, Applicants assert that claim 1 is not anticipated by Ebrahim.

B. Dependent Claim 2

Claim 2 recites in part "deleting…an active doubly linked list that represents one of said groups of integers when all of the integers in said one group are free." (emphasis added) Claim 2 is rejected under the logic:

> "Ebrahim teaches deleting (i.e., removing) (col 8, lines 1 – 27) an active doubly-linked list that represents one of said groups of integers when all of the integers in said one group are free (col 12, lines 30 – 46; see also Figure 8) see also (col 12, lines 30 – 46; see also (col 8, lines 1 – 27; see also Figure 4B))." (Office action page 3)

Applicants have reviewed col. 8, lines 1 – 27, Fig. 4B, col. 12, lines 30 – 46, and Fig. 8 of Ebrahim and have found no reference to "deleting…an active doubly linked list…" as recited in claim 2. Further, Applicants have found no reference to an action that occurs "when all of the integers in said one group are free" as recited in claim 1.

Ebrahim does disclose at col. 8, lines 7 – 9 that "[t]he activation record object reference maps 260 generated by the compiler are stored in the class descriptor 258 portion of the class file." That is, in the alternative embodiment of Fig. 4B, the activation record object reference maps (260) are stored within the class descriptor 258 instead of being stored separately as illustrated in Fig. 4A. Although this alternative embodiment may have the advantage that "the class files are smaller" (col. 8, lines 14 – 15), the active record object reference maps (260) are still stored (e.g., within the class descriptor 258). Given the Ebrahim disclosure, Applicants still assert that Ebrahim does not disclose "deleting…an active doubly linked list that represents one of said groups of integers when all of the integers in said one group are free" as recited in claim 2. Because

Ebrahim does not disclose the "deleting...an active doubly-linked list..." as recited in claim 2, claim 2 is not anticipated by Ebrahim.

C. Dependent Claim 3

Claim 3 includes further limitations related to initializing the doubly linked list from claim 1. Claim 3 is rejected based on the same two references to Ebrahim as claims 1 and 2 (col.8, lines 1 – 27, Fig. 4B, col. 12, lines 30 – 46, and Fig. 8).

As stated above, Applicants assert that Ebrahim does not disclose a doubly linked list. Claim 3 further specifies that each linked list element includes a "next pointer" <u>and</u> a "previous pointer." Applicants assert that Ebrahim does not disclose a doubly linked list in which each linked list element includes a "next pointer" <u>and</u> a "previous pointer" as recited in claim 3. In fact, Applicants have found no mention of a next pointer <u>and</u> a previous pointer in the cited sections of Ebrahim. Additionally, the hash table 208' depicted in Fig. 8 of Ebrahim does not include a doubly linked list in which each element includes <u>a next pointer and a previous pointer</u>. Because Ebrahim does not disclose a doubly linked list that includes a next pointer and a previous pointer as recited in claim 3, claim 3 is not anticipated by Ebrahim.

Claim 3 also recites:

> "forming a doubly linked list, from said linked list elements, <u>that</u> <u>includes all of the integers in said group of integers</u>; and
> establishing <u>a head element having a next pointer</u> for identifying one end of said doubly linked list <u>and a previous pointer</u> for identifying the other end of said doubly linked list." (emphasis added)

Applicants have reviewed the cited references in Ebrahim and assert that Ebrahim does not disclose a doubly linked list "that includes all of the integers in said group of integers" or a "head element having a next pointer...and a previous pointer." In view of this, Applicants assert that claim 3 is not anticipated by Ebrahim.

The above-provided arguments apply also to similar claims of claims 12 – 14 and 33 – 35. Applicants also point out that in claim 12, the doubly linked list is initialized "in response to a request for a next free integer <u>or a specific free integer if said next free integer or said specific free integer is determined to be in a group of integers that has not</u>

been initialized." (emphasis added) Applicants assert that these additional limitations in claim 12 are not disclosed by Ebrahim.
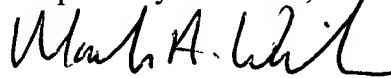
III.   Allowable Subject Matter

Applicants note with appreciation that claims 23 – 32 are allowed and that claims 4 – 11, 15 – 22, and 36 – 43 are objected to as being dependent upon rejected base claims, but would be allowable if rewritten in independent form including all of the limitations of the base claim and any intervening claim. Applicants have not rewritten the objected to claims at this time, as suggested, in view of the above-provided remarks.

Applicants respectfully request reconsideration of the claims in view of the remarks made herein. A notice of allowance is earnestly solicited.

Respectfully submitted,

Date: 5/20/05

Mark A. Wilson
Reg. No. 43,994

Wilson & Ham
PMB: 348
2530 Berryessa Road
San Jose, CA 95132
Phone: (925) 249-1300
Fax: (925) 249-0111

APPENDIX A

for

Application Serial No. 09/909,549

Includes:

doubly linked list, National Institute of Standards and Technology, www.nist.gov (2 pages)

Linked list, http://en.wikipedia.org (14 pages)

NIST

National Institute of Standards and Technology

doubly linked list

(data structure)

Definition: A variant of a *linked list* in which each item has a *link* to the previous item as well as the next. This allows easily accessing list items backward as well as forward and deleting any item in constant time.

Also known as two-way linked list, symmetrically linked list.

See also *jump list*.

Note: See [Stand98, p. 91].

Author: PEB

Implementation

Kaz Kylheku's Kazlib (C) implementing dictionary, dynamic hash table, red-black tree, and doubly linked list.

Go to the Dictionary of Algorithms and Data Structures home page.

If you have suggestions, corrections, or comments, please get in touch with Paul E. Black (paul.black@nist.gov).

Entry modified Fri Dec 17 12:23:42 2004.
HTML page formatted Fri Dec 17 14:50:37 2004.

This page's URL is http://www.nist.gov/dads/HTML/doublyLinkedList.html

NIST
I·C·M·

NIST

National Institute of Standards and Technology

linked list

(data structure)

Definition: A *list* implemented by each item having a *link* to the next item.

Also known as singly linked list.

Specialization (... is a kind of me.)
doubly linked list, *ordered linked list*, *circular list*.

See also *move-to-front heuristic*.

Note: The first item, or <u>head</u>*, is accessed from a fixed location, called a "head pointer." An ordinary linked list must be searched with a* <u>linear search</u>*. Average search time may be improved using a* <u>move-to-front heuristic</u> *in some cases or keeping it an* <u>ordered linked list</u>*. An* <u>external index</u>*, such as a* <u>hash table</u>*,* <u>inverted index</u>*, or auxiliary* <u>search tree</u> *may be used as a "cross index" to help find items quickly.*

A linked list can be used to implement other data structures, such as a <u>queue</u> *or a* <u>stack</u>*.*

Author: PEB

Implementation

explanations, examples, iterators, sorting lists, etc. (C++), Kazlib (C), (C++)

More information

an introduction, a Java applet demonstration.

Go to the Dictionary of Algorithms and Data Structures home page.

If you have suggestions, corrections, or comments, please get in touch with Paul E. Black (paul.black@nist.gov).

Entry modified Fri Dec 17 12:24:13 2004.
HTML page formatted Fri Dec 17 14:50:55 2004.

This page's URL is http://www.nist.gov/dads/HTML/linkedList.html

NIST
ITL

Linked list

From Wikipedia, the free encyclopedia.

In computer science, a **linked list** is one of the fundamental data structures used in computer programming. It consists of a sequence of nodes, each containing arbitrary data fields and one or two references ("links") pointing to the next and/or previous nodes. A linked list is called a self-referential datatype because it contains a pointer or link to another data of the same type. Linked lists permit insertion and removal of nodes at any point in the list in constant time, but do not allow random access. Several different types of linked list exist: singly-linked lists, doubly-linked lists, and circularly-linked lists.

Linked lists can be implemented in most languages. Lisp and Scheme have the data structure built in, along with operations to access the linked list. Languages such as C and C++ rely on pointers and memory addressing to create linked lists.

History

Linked lists were developed in 1955-56 by Allen Newell, Cliff Shaw and Herbert Simon at RAND Corporation as the primary data structure for their Information Processing Language. IPL was used by the authors to develop several early artificial intelligence programs, including the Logic Theory Machine, the General Problem Solver, and a computer chess program. Reports on their work appeared in *IRE Transactions on Information Theory* in 1956, and several conference proceedings from 1957-1959, including *Proceedings of the Western Joint Computer Conference* in 1957 and 1958, and *Information Processing* (Proceedings of the first UNESCO International Conference on Information Processing) in 1959. The now-classic diagram consisting of blocks representing list nodes with arrows pointing to successive list nodes appears in "Programming the Logic Theory Machine" by Newell and Shaw in *Proc. WJCC,* February 1957. Newell and Simon

were recognized with the ACM Turing Award in 1975 for having "made basic contributions to artificial intelligence, the psychology of human cognition, and list processing."

The problem of machine translation for natural language processing led Victor Yngve at MIT to use linked lists as data structures in his COMIT programming language for computer research in the field of linguistics. A report on this language entitled "A programming language for mechanical translation" appeared in *Mechanical Translation* in 1958.

LISP, standing for *list processor*, was created by John McCarthy in 1958 while he was at MIT and in 1960 he published its design in a paper in the Communications of the ACM, entitled "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". One of LISP's major data structures is the linked list.

By the early 1960's, the utility of both linked lists and languages which use these structures as their primary data representation was well established. Bert Green of the MIT Lincoln Laboratory published a review article entitled "Computer languages for symbol manipulation" in *IRE Transactions on Human Factors in Electronics* in March 1961 which summarized the advantages of the linked list approach. A later review article, "A Comparison of list-processing computer languages" by Bobrow and Raphael, appeared in *Communications of the ACM* in April 1964.

Several operating systems developed by Technical Systems Consultants (originally of West Lafayette Indiana, and later of Raleigh, North Carolina) used singly linked lists as file structures. A directory entry pointed to the first sector of a file, and succeeding portions of the file were located by traversing pointers. Systems using this technique included Flex (for the Motorola 6800 CPU), mini-Flex (same CPU), and Flex9 (for the Motorola 6809 CPU). A variant developed by TSC for and marketed by Smoke Signal Broadcasting in California, used doubly linked lists in the same manner.

The TSS operating system, developed by IBM for the System 360/370 machines, used a double linked list for their file system catalog. The directory structure was similar to Unix, where a directory could contain files and/or other directories and extend to any depth. A utility *flea* was created to fix file system problems after a crash, since modified portions of the file catalog were sometimes in memory when a crash occurred. Problems were detected by comparing the forward and backward links for consistency. If a forward link was corrupt, then if a backward link to the infected node was found, the forward link was set to the node with the backward link. A humorous comment in the source code where this utility was invoked stated "Everyone knows a flea caller gets rid of bugs in cats."

Variants

Singly-linked list

The simplest kind of linked list is a *singly-linked list* (or *slist* for short), which has one link per node. This link points to the next node in the list, or to a null value or empty list if it is the last node.

A singly linked list containing three integer values

Doubly-linked list

A more sophisticated kind of linked list is a *doubly-linked list*. Each node has two links, one to the previous node and one to the next.
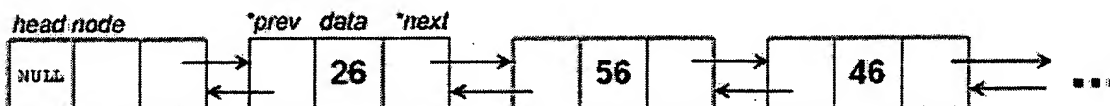
An example of a doubly linked list.

In some very low level languages, Xor-linking offers a way to implement doubly-linked lists using a single word for both links, although the use of this technique is usually discouraged.

Circularly-linked list

In a *circularly-linked list*, the first and last nodes are linked together. This can be done for both singly and doubly linked lists. To traverse a circular linked list, you begin at any node and follow the list in either direction until you return to the original node. Viewed another way, circularly-linked lists can be seen as having no beginning or end. This type of list is most useful for managing buffers for data ingest, and in cases where you have one object in a list and wish to see all other objects in the list.

Sentinel nodes

Linked lists sometimes have a special *dummy* or *sentinel node* at the beginning and/or at the end of the list, which is not used to store data. Its purpose is to simplify or speed up some operations, by ensuring that every data node always has a previous and/or next node, and that every list (even one that contains no data elements) always has a "first" and "last" node.

Applications of linked lists

Linked lists are used as a building block for many other data structures, such as stacks, queues and their variations.

The "data" field of a node can be another linked list. By this device, one can construct many linked data structures with lists; this practice originated in the Lisp programming language, where linked lists are a primary (though by no means the only) data structure, and is now a common feature of the functional programming style.

Sometimes, linked lists are used to implement associative arrays, and are in this context called **association lists**. There is very little good to be said about this use of linked lists; they are easily outperformed by other data structures such as self-balancing binary search trees even on small data sets (see the discussion in associative array). However, sometimes a linked list is dynamically created out of a subset of nodes in such a tree, and used to more efficiently traverse that set.

Tradeoffs

As with most choices in computer programming and design, no method is well suited to all circumstances. A linked list data structure might work well in one case, but cause problems in another. This is a list of some of the common tradeoffs involving linked list structures. In general, if you have a dynamic collection, where elements are frequently being added and deleted, and the location of new elements added to the list is significant, then benefits of a linked list increase.

Linked lists vs. arrays

Linked lists have several advantages over arrays. They allow a new element to be inserted or deleted at any position in a constant number of operations (changing some references), while arrays require a linear ($O(n)$) number of operations. Elements can also be inserted into linked lists indefinitely, while an array will eventually either fill up or need to be resized, an expensive operation that may not even be possible if memory is fragmented. Similarly, an array from which many elements are removed may become wastefully empty or need to be made smaller.

	Array	Linked list
Indexing	$O(1)$	$O(n)$
Inserting	$O(n)$	$O(1)$
Deleting	$O(n)$	$O(1)$
Persistent	No	Singly yes

Further memory savings can be achieved, in certain cases, by sharing the same "tail" of elements among two or more lists — that is, the lists end in the same sequence of elements. In this way, one can add new elements to the front of the list while keeping a reference to both the new and the old versions — a simple example of a persistent data structure.

Locality	Great	Bad

On the other hand, arrays allow random access, while linked lists allow only sequential access to elements. Singly-linked lists, in fact, can only be traversed in one direction. This makes linked lists unsuitable for applications where it's useful to look up an element by its index quickly, such as heapsort. Sequential access on arrays is also faster than on linked lists on many machines due to locality of reference and data caches. Linked lists receive almost no benefit from the cache.

Another disadvantage of linked lists is the extra storage needed for references, which often makes them impractical for lists of small data items such as characters or boolean values. It can also be slow, and with a naïve allocator, wasteful, to allocate memory separately for each new element, a problem generally solved using memory pools.

A number of linked list variants exist that aim to ameliorate some of the above problems. Unrolled linked lists store several elements in each list node, increasing cache performance while decreasing memory overhead for references. CDR coding does both these as well, by replacing references with the actual data referenced, which extends off the end of the referencing record.

A good example that highlights the pros and cons of using arrays vs. linked lists is by implementing a program that resolves the Josephus problem. The Josephus problem is an election method that works by having a group of people stand in a circle. Starting at a predetermined person, you count around the circle n times. Once you reach nth person, take them out of the circle and have the members close the circle. Then count around the circle the same n times and repeat the process, until only one person is left. That person wins the election. This shows the strengths and weaknesses of a linked list vs. an array, because if you view the people as connected nodes in a circular linked list then it shows how easily the linked list is able to delete nodes (as it only has to rearrange the links to the different nodes). However, the linked list will be poor at finding the next person to remove and will need to recurse through the list till it finds that person. An array, on the other hand, will be poor at deleting nodes (or elements) as it cannot remove one node without individually shifting all the elements up the list by one. However, it is exceptionally easy to find the nth person in the circle by directly referencing them by their position in the array.

Double-linking vs. single-linking

Double-linked lists require more space per node (unless one uses xor-linking), and their elementary operations are more expensive; but they are often easier to manipulate because they allow sequential access to the list in both directions. In particular, one can insert or delete a node in a constant number of operations given only that node's address. Some algorithms require access in both directions. On the other hand, they do not allow tail-sharing, and cannot be used as persistent data structures.

Circular lists

Circular linked lists are most useful for describing naturally circular structures, and have the advantage of regular structure and being able to traverse the list starting at any point. They also allow quick access to the first and last records through a single pointer (the address of the last element). Their main disadvantage is the complexity of iteration, which has subtle special cases.

Sentinel nodes

Sentinel nodes may simplify certain list operations, by ensuring that the next and/or previous nodes exist for every

element. However sentinel nodes use up extra space (especially in applications that use many short lists), and they may complicate other operations. To avoid the extra space requirement the sentinel nodes can often be reused as head and/or tail pointers.

Linked list operations

When manipulating linked lists in-place, care must be taken to not use values that you have invalidated in previous assignments. This makes algorithms for inserting or deleting linked list nodes somewhat subtle. This section gives pseudocode for adding or removing nodes from singly, doubly, and circularly linked lists in-place. Throughout we will use *null* to refer to an end-of-list marker or sentinel, which may be implemented in a number of ways.

Singly-linked lists

Our node data structure will have two fields. We also keep a variable *head* which always points to the first node in the list, or is *null* for an empty list.

```
record Node {
    data // The data being stored in the node;
         // often a reference to the actual data
    next // A reference to the next node
}

Node head  // points to front of list
Node nodeBeingRemoved  // used in later examples
```

Traversal of a singly-linked list is easy, beginning at the first node and following each *next* link until we come to the end:

```
node := head // start at front of list
while node not null { // loop through list
    (do something with node.data)
    node := node.next // go to next node in list
}
```

The following code inserts a node after an existing node in a singly linked list. The diagram shows how it works. Inserting a node before an existing one cannot be done; instead, you have to locate it while keeping track of the previous node.
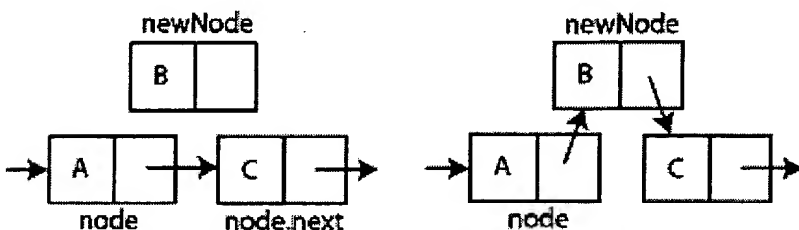


```
function insertAfter(Node node, Node newNode) { // insert newNode after node
    newNode.next := node.next // point new node to next node
    node.next    := newNode    // add new node to list
}
```

Inserting at the beginning of the list requires a separate function. This requires updating *head*.

```
function insertBeginning(Node newNode) { // insert node before current head
    newNode.next := head     // point to rest of list
    head           := newNode // new front of list
}
```

Similarly, we have functions for removing the node *after* a given node, and for removing a node from the beginning of the list. The diagram demonstrates the former. To find and remove a particular node, one must again keep track of the previous element.
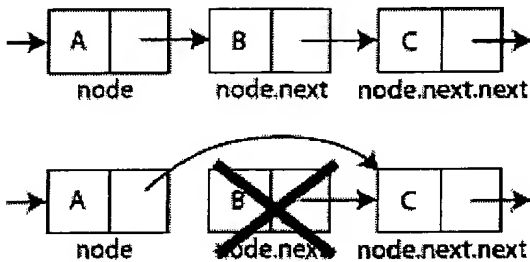

```
function removeAfter(Node node) { // remove node past this one
    nodeBeingRemoved := node.next     // this is the node to delete
    node.next := node.next.next       // point past node
    (the above statement could be node.next = nodeBeingRemoved.next)
    destroy nodeBeingRemoved           // free up deleted node
}
```

```
function removeBeginning(Node node) { // remove head node
    nodeBeingRemoved := head     // this is the node to delete
    head := head.next            // point past deleted node
    destroy nodeBeingRemoved     // free up deleted node
}
```

Notice that removeBeginning() sets *head* to *null* when removing the last node in the list.

Doubly-linked lists

With doubly-linked lists there are even more pointers to update, but also less information is needed, since we can use backwards pointers to observe preceding elements in the list. This enables new operations, and eliminates special-case functions. We will add a *prev* field to our nodes, pointing to the previous element, and a *tail* variable which always points to the last node in the list. Both *head* and *tail* are *null* for an empty list.

Iterating through a doubly linked list can be done in either direction. In fact, direction can change many times, if desired.

Forwards

```
node := head
while node ≠ null
    <do something with node.data>
    node := node.next
```
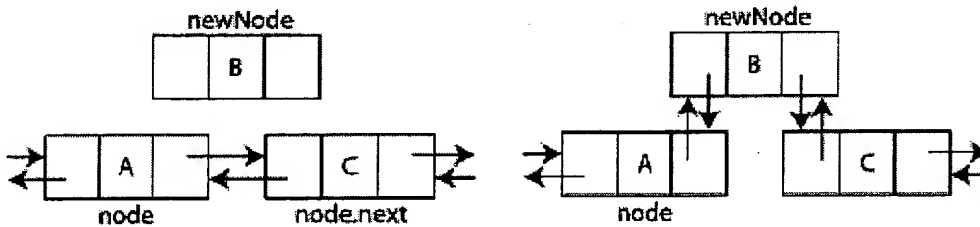
Backwards

```
node := tail
while node ≠ null
    <do something with node.data>
    node := node.prev
```

These symmetric functions add a node either after or before a given node, with the diagram demonstrating after:


```
function insertAfter(node, newNode)
    newNode.prev := node
    newNode.next := node.next
    if node.next = null
        tail := newNode
    else
        node.next.prev := newNode
    node.next := newNode
```

```
function insertBefore(node, newNode)
    newNode.prev := node.prev
    newNode.next := node
    if node.prev is null
        head := newNode
    else
        node.prev.next := newNode
    node.prev    := newNode
```

We also need a function to insert a node at the beginning of a possibly-empty list:

```
function insertBeginning(newNode)
    if head = null
        head := newNode
        tail := newNode
        newNode.prev := null
        newNode.next := null
    else
        insertBefore(head, newNode)
```

A symmetric function inserts at the end:

```
function insertEnd(newNode)
    if tail = null
        insertBeginning(newNode)
    else
        insertAfter(tail, newNode)
```

Removing a node is easier, only requiring care with *head* and *tail*:

```
function remove(node)
   if node.prev = null
       head = node.next
   else
       node.prev.next = node.next

   if node.next = null
       tail := node.prev
   else
       node.next.prev = node.prev
   destroy node
```

One subtle consequence of this procedure is that deleting the last element of a list sets both *head* and *tail* to *null*.

Circularly-linked lists

Circularly-linked lists can be either singly or doubly linked. In a circularly linked list, all nodes are linked in a continuous circle, without using *null*. For lists with a front and a back (such as a queue), one stores a reference to the last node in the list. The *next* node after the last node is the first node. Elements can be added to the back of the list and removed from the front in constant time.

Both types of circularly-linked lists benefit from the ability to traverse the full list beginning at any given node. This often allows us to avoid storing *head* and *tail*, although if the list may be empty we need a special representation for the empty list, such as a *head* variable which points to some node in the list or is *null* if it's empty; we use such a *head* here. This representation significantly simplifies adding and removing nodes with a non-empty list, but empty lists are then a special case.

Assuming that *someNode* is some node in a non-empty list, this code iterates through that list starting with *someNode*:

Forwards

```
node := someNode
do
    do something with node.value
    node := node.next
while node ≠ someNode
```

Backwards

```
node := someNode
do
    do something with node.value
    node := node.prev
while node ≠ someNode
```

Notice the postponing of the test to the end of the loop. This is important for the case where the list contains only the single node *someNode*.

This simple function inserts a node into a doubly-linked circularly linked list after a given element:

```
function insertAfter(node, newNode)
    newNode.next := node.next
    newNode.prev := node
    node.next.prev := newNode
```

```
        node.next       := newNode
```

Inserting an element in a possibly empty list requires a special function:

```
function insertBeginning(node)
    if head = null
        node.prev := node
        node.next := node
    else
        insertAfter(head.prev, node)
    head := node
```

Finally, removing a node must deal with the case where the list empties:

```
function remove(node)
    if node.next = node
        head := null
    else
        node.next.prev := node.prev
        node.prev.next := node.next
    destroy node
```

Linked lists using arrays of nodes

Languages that do not support any type of reference can still create links by replacing pointers with array indices. The approach is to keep an array of records, where each record has integer fields indicating the index of the next (and possibly previous) node in the array. Not all nodes in the array need be used. If even records are not supported, parallel arrays can often be used instead.

As an example, consider the following linked list record that uses arrays instead of pointers:

```
record Entry {
    integer next; // location of next entry in array
    integer prev; // previous entry (if double-linked)
    string Name;  // Name on account                                    /
    real balance; // Account balance
}
```

By creating an array of these structures, and an integer variable to store the index of the head element, a linked list can be built:

```
integer listHead;
Entry Records[1000];
```

Links between elements are formed by placing the array index of the next (or previous) cell into the Next or Prev field within a given element. For example:

Index	Next	Prev	Name	Balance
0	1	4	Jones, John	123.45
1	-1	0	Smith, Joseph	234.56

2	4	-1	Adams, Adam	0.00
3			Ignore, Ignatius	999.99
4	0	2	Another, Anita	876.54
5				
6				
7				

In the above example, `ListHead` would be set to 4, the location of the first entry in the list. Notice that entry 3 and 5 through 7 are not part of the list. These cells are available for any additions to the list. By creating a `ListFree` integer variable, a free list could be created to keep track of what cells are available. If all entries are in use, the size of the array would have to be increased or some elements would have to be deleted before new entries can be stored in the list.

The following code would traverse the list and display names and account balance:

```
i := listHead; // start at beginning of list
while i >= 0 { '// loop through the list
    print i, Records[i].name, Records[i].balance // print entry
    i = Records[i].next; // next entry in list
}
```

When faced with a choice, the advantages of this approach include:

- The linked list is relocatable, meaning it can be moved about in memory at will, and it can also be quickly and directly serialized for storage on disk or transfer over a network.
- Especially for a small list, array indexes can occupy significantly less space than a full pointer on many architectures.
- Locality of reference can be improved by keeping the nodes together in memory and by periodically rearranging them, although this can also be done in a general store.
- Naïve dynamic memory allocators can produce an excessive amount of overhead storage for each node allocated; almost no allocation overhead is incurred per node in this approach.
- Seizing an entry from a pre-allocated array is faster than using dynamic memory allocation for each node, since dynamic memory allocation typically requires a search for a free memory block of the desired size.

This approach has one main disadvantage, however: it creates and manages a private memory space for its nodes. Not only does this increase complexity of the implementation, but growing it may be difficult or impossible, because it is large, whereas finding space for a new linked list node in a large, general memory pool is easier. This slow growth also affects algorithmic performance, as it causes a few insert operations to unexpectedly take linear (O(n)) instead of constant time (although it's still amortized constant). Using a general memory pool also leaves more memory for other data if the list is smaller than expected or if many nodes are freed. However, since this approach is mainly used for languages that do not support dynamic memory allocation, the block of memory is created at compile / load time, this is usually not an issue. These disadvantages are also mitigated if the maximum size of the list is known at the time the array is created.

Language support

Many programming languages such as Lisp and Scheme have singly linked lists built in. In many functional languages, these lists are constructed from nodes, each called a *cons* or *cons cell*. The cons has two fields: the *car*, a reference to the data for that node, and the *cdr*, a reference to the next node. Although cons cells can be used to build other data structures, this is their primary purpose.

In other languages, linked list are typically built using references together with records. Here is a complete example in C:

```c
#include <stdio.h>   /* for printf */
#include <stdlib.h>  /* for malloc */

typedef struct ns {
        int data;
        struct ns *next;
} node;

node *list_add(node ** p, int i) {
    node *n = malloc(sizeof(*n));
    n->next = *p;
    *p = n;
    n->data = i;
    return n;
}

void list_remove(node ** p) {
    if (p) {
        node *n = *p;
        *p = (*p)->next;
        free(n);
    }
}

node **list_search(node ** n, int i) {
    if (!n) return NULL;
    while (*n) {
        if ((*n)->data == i) {
            return n;
        }
    n = &(*n)->next;
    }
    return NULL;
}

void list_print(node * n) {
    if (!n) printf("list is empty\n");
    while (n) {
        printf("print %p %p %i \n", n, n->next, n->data);
        n = n->next;
    }
}

int main(void) {
    node *n = NULL;

    list_add(&n, 0);
    list_add(&n, 1);
    list_add(&n, 2);
    list_add(&n, 3);
    list_add(&n, 4);
    list_print(n);
    list_remove(&n);                /* remove head */
    list_remove(&n->next);          /* remove second */
    list_remove(list_search(&n, 1));
    list_remove(&n->next);          /* remove tail */
    list_remove(&n);                /* remove last */
    list_print(n);

    return 0;
}
```

Internal and external storage

When constructing a linked list, one is faced with the choice of whether to store the data of the list directly in the linked list nodes, called *internal storage*, or to merely store a reference to the data, called *external storage*. Internal storage has the advantage of making access to the data more efficient, requiring less storage overall, having better locality of reference, and simplifying memory management for the list (its data is allocated and deallocated at the same time as the

list nodes).

External storage, on the other hand, has the advantage of being more generic, in that the same data structure and machine code can be used for a linked list no matter what the size of the data is. It also makes it easy to place the same data in multiple linked lists. Note, however, that even with internal storage the same data can be placed in multiple lists, either by including multiple *next* references in the node data structure, or by having other linked lists store references to the nodes of the linked list containing the data.

A common misconception is that every list using internal storage requires a new set of functions to operate on the linked list, consuming more effort and code space than an approach using external storage. This is true with a naïve approach, but as long as the *next* pointers are located in the same place in each record, it is possible to create a generic implementation of linked lists with internal storage that only needs to be supplied for each type of list a way of creating and destroying nodes.

Example

Suppose you wanted to create a linked list of families and their members. Using internal storage, the structure might look like the following:

```
record member { // structure for a member of a family
    member next // pointer to other members of family
    string fname // first name
    integer age // age of member
}
record family { // structure for the family itself
    family next // pointer to other families
    string lname // last name of family
    string address // address of family
    string phone // phone number of family
    member members // pointer to list of members of family
}
```

To print a complete list of families and their members using internal storage, we could write:

```
aFamily := Families // start at head of list
while aFamily ≠ null { // loop through list of families
    print information about family
    aMember := aFamily.members // get list of family members
    while aMember ≠ null { // loop through list of members
        print information about member
        aMember := aMember.next // go to next member
    }
    aFamily := aFamily.next // go to next family
}
```

Using external storage, we would create the following structures:

```
record node { // generic link structure
    node next // pointer to next node
    pointer data // generic pointer for data at node
}
record member { // structure for family member
    string fname // first name
    integer age // age of member
}
record family { // structure for family
    string lname // last name of family
    string address // address of family
```

```
        string phone // phone number of family
        node members // pointer to list of members of family
}
```

To print a complete list of families and their members using external storage, we could write:

```
famNode := Families // start at head of list
while famNode ≠ null { // loop through list of families
    aFamily = (family)famNode.data // extract family from node
    print information about family
    memNode := aFamily.members // get list of family members
    while memNode ≠ null { // loop through list of members
        aMember := (member)memNode.data // extract member from node
        print information about member
        memNode := memNode.next // get next family member
    }
    famNode := famNode.next // get next family in list
}
```

Notice that when using external storage, an extra step is needed to extract the record from the node and cast it into the proper data type.

Related data structures

The skip list is a linked list augmented with layers of pointers for quickly jumping over large numbers of elements, and then descending to the next layer. This process continues down to the bottom layer, which is the actual list.

A binary tree can be seen as a type of linked list where the elements are themselves linked lists of the same nature. The result is that each node may include a reference to the head node of one or two other linked lists, which, together with their contents, form the subtrees below that node.

An unrolled linked list is a linked list in which each node contains an array of data values. This leads to improved cache performance, since more list elements are contiguous in memory, and reduced memory overhead, because less metadata needs to be stored for each element of the list.

References

- National Institute of Standards and Technology (August 16, 2004). Definition of a linked list (*http://nist.gov/dads/HTML/linkedList.html*). Retrieved December 14, 2004.
- Antonakos, James L. and Mansfield, Kenneth C., Jr. *Practical Data Structures Using C/C++* (1999). Prentice-Hall. ISBN 0-13-280843-9, pp. 165–190
- Collins, William J. *Data Structures and the Java Collections Framework* (2002,2005) New York, NY: McGraw Hill. ISBN 0-07-282379-8, pp. 239–303
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford *Introductions to Algorithms* (2003). MIT Press. ISBN 0-262-03293-7, pp. 205–213, 501–505
- Green, Bert F. Jr. (1961). Computer Languages for Symbol Manipulation. *IRE Transactions on Human Factors in Electronics.* 2 pp. 3-8.
- McCarthy, John (1960). Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM.* [1] (*http://www-formal.stanford.edu/jmc/recursive.html*) HTML (*http://www-formal.stanford.edu/jmc/recursive/recursive.html*) DVI (*http://www-formal.stanford.edu/jmc/recursive.dvi*) PDF (*http://www-formal.stanford.edu/jmc/recursive.pdf*) PostScript (*http://www-formal.stanford.edu/jmc/recursive.ps*)
- Newell, Allen and Shaw, F. C. (1957). Programming the Logic Theory Machine. *Proceedings of the Western Joint Computer Conference.* pp. 230-240.
- Parlante, Nick (2001). Linked list basics. *Stanford University.* PDF

(*http://cslibrary.stanford.edu/103/LinkedListBasics.pdf*)
- Sedgewick, Robert *Algorithms in C* (1998). Addison Wesley. ISBN 0-201-31452-5, pp. 90–109
- Shaffer, Clifford A. *A Practical Introduction to Data Structures and Algorithm Analysis* (1998). NJ: Prentice Hall. ISBN 0-13-660911-2, pp. 77–102
- Wilkes, Maurice Vincent (1964). An Experiment with a Self-compiling Compiler for a Simple List-Processing Language. *Annual Review in Automatic Programming* **4**, 1. Published by Pergamon Press.
- Wilkes, Maurice Vincent (1964). Lists and Why They are Useful. *Proceeds of the ACM National Conference, Philadelphia 1964* (ACM Publication P-64 page F1-1); Also *Computer Journal* **7**, 278 (1965).
- Kulesh Shanmugasundaram (April 4, 2005). Linux Kernel Linked List Explained (*http://isis.poly.edu/kulesh/stuff/src/klist/*).

External links

- Description (*http://nist.gov/dads/HTML/linkedList.html*) from the Dictionary of Algorithms and Data Structures
- Some linked list materials are available from the Stanford University Computer Science department:
    - Introduction to Linked Lists (*http://cslibrary.stanford.edu/103/*)
    - Linked List Problems (*http://cslibrary.stanford.edu/105/*)
- Citations from CiteSeer (*http://citeseer.ist.psu.edu/cis?q=linked+and+list+and+data+and+structure*)

Retrieved from "http://en.wikipedia.org/wiki/Linked_list"

Categories: Data structures

- This page was last modified 07:20, 5 May 2005.
- All text is available under the terms of the GNU Free Documentation License (see **Copyrights** for details).